组号: 6

图片包含 游戏机, 画

描述已自动生成

上海大学计算机工程与科学学院

**实 验 报 告**

（数据结构1）

学 期： 大二冬季学期

组 长： 饶思莹

学 号： 23121617

指导教师： 朱频频

成绩评定： （教师填写）

二〇二三年四月六日

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **小组信息** | | | | |
| 登记序号 | 姓名 | 学号 | 分工 | 签名 |
| 1 | 林仪 | 23121029 | ppt制作 |  |
| 2 | 罗苗 | 23121791 | 报告 |  |
| 3 | 彭欣然 | 23122809 | 代码 |  |
| 4 | 顾宜凌 | 23121721 | 代码 |  |
| 5 | 张静雯 | 23121790 | 代码及整合、汇报 |  |
| 6 | 饶思莹 | 23121617 | 代码及整合、汇报 |  |
| 7 | 杨源 | 23121651 | 代码 |  |
| 8 | 李临宜 | 23122814 | 代码 |  |

|  |  |
| --- | --- |
| **实验概述** | |
| 实验零 | （熟悉上机环境、进度安排、评分制度；确定小组成员） |
| 实验一 |  |
| 实验二 |  |
| 实验三 |  |
| 实验四 |  |

# 实验二

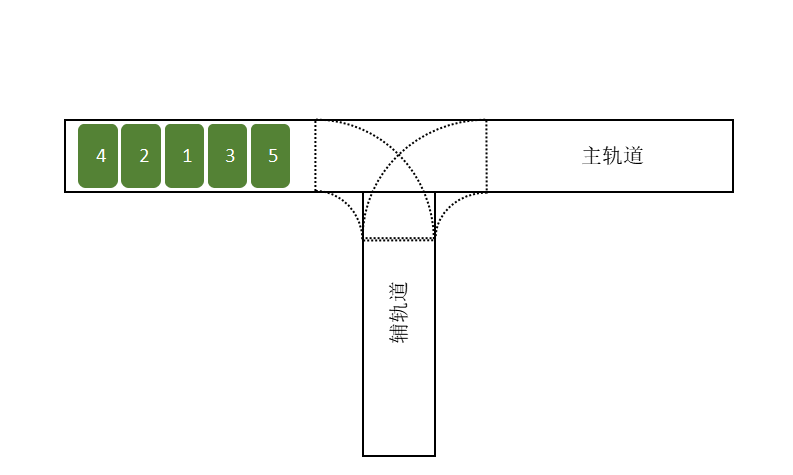
**一、实验题目**

“丁”字型铁路调度系统的实现

**二、实验内容**

**1、任务目标**

  “丁”字型铁路调度系统如下图所示，由垂直的两条铁轨组成，水平方向的为主铁轨，竖直方向的为辅助铁轨。辅助铁轨用于对车厢次序进行调整，它在主铁轨中间，把主铁轨分成左右两部分。主铁轨左边的车厢只能从左边开到右边；或者从主铁轨左边进入辅助铁轨；辅助铁轨上的车厢只可以进入主铁轨右边。



现在有n节车厢，编号为1、2、......、n，在主铁轨的左边以随机的顺序排列，要求通过这个调度系统，在主铁轨的右边顺序开出（即按1、2、...、n的顺序）。

基于栈和队列的相关知识和代码实现，实现该铁路调度系统对车厢的调度过程。

**2、任务描述**

1）可以在教材代码的基础上进行编码，也可以自行从头开始编码；

2）可以手动输入待调度的车厢初始序列（编号之间用空格或换行分开），输出是否能调度成功，以及调度每一步的调度过程；并输出栈的总入栈（出栈）次数；输出调度过程中所需辅助轨道的最大长度。

3）当n=100时，随机生成100个初始序列，测试这些初始序列能否被成功调度，并输出；

4）分析使用队列作为辅助轨道（尾进头出），在相同情况下与栈辅助轨道的区别；

5）提供菜单，实现以上功能。

**三、解决方案**

**1、算法设计**

在本实验中使用C++语言进行编程，根据实验要求：

（1）设计了一个 通用异常类（Error），用于提示异常，该类位于头文件Assistance.h，是后续头文件以及源文件运行的基础。

（2）设计了一个顺序栈模板类(SeqStack)，提供了基本的栈操作，包括创建、销毁、判断空、清空、入栈、出栈和遍历等，适用于需要栈结构的多种场景。

（3）设计了一个循环队列模板类(SeqQueue),提供了基本的队列操作，包括创建、销毁、判断空、清空、入队、出队和遍历等。它允许以线性但循环的方式存储元素，它的最后一个物理位置连接到第一个物理位置，形成一个环状结构，这有助于更高效地利用固定的内存空间。

（4）利用SMFL多媒体库的2D图形模块设计了菜单，界面美观，交互性强。

|  |
| --- |
| 头文件Assistance.h |
| #pragma once  #pragma once  #ifndef \_\_ASSISTANCE\_H\_\_ // 如果没有定义\_\_ASSISTANCE\_H\_\_  #define \_\_ASSISTANCE\_H\_\_ // 那么定义\_\_ASSISTANCE\_H\_\_  // 辅助软件包  // ANSI C++标准库头文件  #include <cstring> // 标准串操作  #include <iostream> // 标准流操作  #include <limits> // 极限  #include <cmath> // 数据函数  #include <fstream> // 文件输入输出  #include <cctype> // 字符处理  #include <ctime> // 日期和时间函数  #include <cstdlib> // 标准库  #include <cstdio> // 标准输入输出  #include <iomanip> // 输入输出流格式设置  #include <cstdarg> // 支持变长函数参数  #include <cassert> // 支持断言  using namespace std; // 标准库包含在命名空间std中  // 自定义类型  enum Status {  SUCCESS, FAIL, UNDER\_FLOW, OVER\_FLOW, RANGE\_ERROR, DUPLICATE\_ERROR,  NOT\_PRESENT, ENTRY\_INSERTED, ENTRY\_FOUND, VISITED, UNVISITED  };  // 宏定义  #define DEFAULT\_SIZE 1000 // 缺省元素个数  #define DEFAULT\_INFINITY 1000000 // 缺省无穷大  // 辅助函数声明  char GetChar(istream& inStream = cin); // 从输入流inStream中跳过空格及制表符获取一字符  template <class ElemType >  void Swap(ElemType& e1, ElemType& e2); // 交换e1, e2之值  template<class ElemType>  void Display(ElemType elem[], int n); // 显示数组elem的各数据元素值  template <class ElemType>  void Write(const ElemType& e); // 显示数据元素  // 辅助类  class Error; // 通用异常类  char GetChar(istream& inStream)  // 操作结果：从输入流inStream中跳过空格及制表符获取一字符  {  char ch; // 临时变量  while ((ch = (inStream).peek()) != EOF // 文件结束符(peek()函数从输入流中接受1  // 字符,流的当前位置不变)  && ((ch = (inStream).get()) == ' ' // 空格(get()函数从输入流中接受1字符,流  // 的当前位置向后移1个位置)  || ch == '\t')); // 制表符  return ch; // 返回字符  }  // 通用异常类  #define MAX\_ERROR\_MESSAGE\_LEN 100  class Error  {  private:  // 数据成员  char message[MAX\_ERROR\_MESSAGE\_LEN];// 异常信息  public:  // 方法声明  Error(const char\* mes = "一般性异常!"); // 构造函数  ~Error(void) {}; // 析构函数  void Show() const; // 显示异常信息  };  // 通用异常类的实现部分  Error::Error(const char\* mes) {  strcpy\_s(message, sizeof(message), mes); // 安全复制异常信息  }  void Error::Show()const  // 操作结果：显示异常信息  {  cout << message << endl; // 显示异常信息  }  template <class ElemType >  void Swap(ElemType& e1, ElemType& e2)  // 操作结果: 交换e1, e2之值  {  ElemType temp; // 临时变量  // 循环赋值实现交换e1, e2  temp = e1; e1 = e2; e2 = temp;  }  template<class ElemType>  void Display(ElemType elem[], int n)  // 操作结果: 显示数组elem的各数据元素值  {  for (int i = 0; i < n; i++)  { // 显示数组elem  cout << elem[i] << " ";  }  cout << endl;  }  template <class ElemType>  void Write(const ElemType& e)  // 操作结果: 显示数据元素  {  cout << e << " ";  }  #endif |
| 头文件SeqStack.h |
| #pragma once  #ifndef \_\_SQ\_STACK\_H\_\_  #define \_\_SQ\_STACK\_H\_\_  #include <functional> // 包含 std::function  // 定义 Status 类型（如果尚未定义）  // enum Status {  // SUCCESS,  // OVER\_FLOW,  // UNDER\_FLOW  // };  // 顺序栈模板类  template<class ElemType>  class SeqStack {  protected:  // 顺序栈的数据成员:  int top; // 栈顶指针  int maxSize; // 栈的最大容量  ElemType\* elems; // 元素存储空间  public:  // 顺序栈的方法声明及重载编译系统默认方法声明:  SeqStack(int size = DEFAULT\_SIZE); // 构造函数  virtual ~SeqStack(); // 析构函数  int GetLength() const; // 求栈的长度  bool IsEmpty() const; // 判断栈是否为空  void Clear(); // 将栈清空  void Traverse(const std::function<void(const ElemType&)>& Visit) const; // 遍历栈  Status Push(const ElemType e); // 入栈  Status Top(ElemType& e) const; // 取顶元素  Status Pop(ElemType& e); // 出栈  SeqStack(const SeqStack<ElemType>& s); // 复制构造函数  SeqStack<ElemType>& operator =(const SeqStack<ElemType>& s); // 赋值语句重载  };  // 顺序栈类的实现部分  template<class ElemType>  SeqStack<ElemType>::SeqStack(int size)  // 操作结果：构造一个最大容量为size的空栈  {  maxSize = size; // 栈的最大容量  elems = new ElemType[maxSize]; // 分配新的存储空间  top = -1;  }  template<class ElemType>  SeqStack<ElemType>::~SeqStack()  // 操作结果：销毁栈  {  delete[]elems; // 释放栈的存储空间  }  template <class ElemType>  int SeqStack<ElemType>::GetLength() const  // 操作结果：返回栈中元素个数  {  return top + 1;  }  template<class ElemType>  bool SeqStack<ElemType>::IsEmpty() const  // 操作结果：如栈为空，则返回true，否则返回false  {  return top == -1;  }  template<class ElemType>  void SeqStack<ElemType>::Clear()  // 操作结果：清空栈  {  top = -1;  }  template <class ElemType>  void SeqStack<ElemType>::Traverse(const std::function<void(const ElemType&)>& Visit) const {  for (int i = top; i >= 0; i--) {  Visit(elems[i]);  }  }  template<class ElemType>  Status SeqStack<ElemType>::Push(const ElemType e)  // 操作结果：将元素e追加到栈顶,如成功则返加SUCCESS,如栈已满将返回OVER\_FLOW  {  if (top == maxSize - 1) // 栈已满  return OVER\_FLOW;  else { // 操作成功  elems[++top] = e; // 将元素e追加到栈顶  return SUCCESS;  }  }  template<class ElemType>  Status SeqStack<ElemType>::Top(ElemType& e) const  // 操作结果：如栈非空,用e返回栈顶元素,函数返回SUCCESS,否则函数返回UNDER\_FLOW  {  if (IsEmpty()) // 栈空  return UNDER\_FLOW;  else { // 栈非空,操作成功  e = elems[top]; // 用e返回栈顶元素  return SUCCESS;  }  }  template<class ElemType>  Status SeqStack<ElemType>::Pop(ElemType& e)  // 操作结果：如栈非空,删除栈顶元素,并用e返回栈顶元素,函数返回SUCCESS,否则  // 函数返回UNDER\_FLOW  {  if (IsEmpty()) // 栈空  return UNDER\_FLOW;  else { // 操作成功  e = elems[top--]; // 用e返回栈顶元素  return SUCCESS;  }  }  template<class ElemType>  SeqStack<ElemType>::SeqStack(const SeqStack<ElemType>& s)  // 操作结果：由栈s构造新栈--复制构造函数  {  maxSize = s.maxSize; // 栈的最大容量  elems = new ElemType[maxSize]; // 分配存储空间  top = s.top; // 复制栈顶指针  for (int i = 0; i <= top; i++) // 从栈底到栈顶对栈s的每个元素进行复制  elems[i] = s.elems[i];  }  template<class ElemType>  SeqStack<ElemType>& SeqStack<ElemType>::operator =(const SeqStack<ElemType>& s)  // 操作结果：将栈s赋值给当前栈--赋值语句重载  {  if (&s != this) {  if (elems != NULL) delete[]elems; // 释放已有存储空间  maxSize = s.maxSize; // 栈的最大容量  elems = new ElemType[maxSize]; // 分配存储空间  top = s.top; // 复制栈顶指针  for (int i = 0; i <= top; i++) // 从栈底到栈顶对栈s的每个元素进行复制  elems[i] = s.elems[i];  }  return \*this;  }  #endif // \_\_SQ\_STACK\_H\_\_ |
| 头文件SeqQueue.h |
| #pragma once  #ifndef \_\_SQ\_QUEUE\_H\_\_  #define \_\_SQ\_QUEUE\_H\_\_  #include <functional> // 包含 std::function  // 定义 Status 类型（如果尚未定义）  // enum Status {  // SUCCESS,  // OVER\_FLOW,  // UNDER\_FLOW  // };  // 循环队列模板类  template<class ElemType>  class SeqQueue {  protected:  // 循环队列的数据成员:  int front; // 队头指针  int rear; // 队尾指针  int maxSize; // 队列的最大容量  ElemType\* elems; // 元素存储空间  public:  // 循环队列的方法声明及重载编译系统默认方法声明:  SeqQueue(int size = DEFAULT\_SIZE); // 构造函数  virtual ~SeqQueue(); // 析构函数  int GetLength() const; // 求队列的长度  bool IsEmpty() const; // 判断队列是否为空  void Clear(); // 将队列清空  void Traverse(const std::function<void(const ElemType&)>& Visit) const; // 遍历队列  Status EnQueue(const ElemType e); // 入队  Status DelQueue(ElemType& e); // 出队  Status GetHead(ElemType& e) const; // 取队首元素  SeqQueue(const SeqQueue<ElemType>& s); // 复制构造函数  SeqQueue<ElemType>& operator =(const SeqQueue<ElemType>& s); // 赋值语句重载  };  // 循环队列类的实现部分  template<class ElemType>  SeqQueue<ElemType>::SeqQueue(int size)  // 操作结果：构造一个最大容量为size的空队列  {  maxSize = size; // 队列的最大容量  elems = new ElemType[maxSize]; // 分配新的存储空间  front = 0;  rear = 0;  }  template<class ElemType>  SeqQueue<ElemType>::~SeqQueue()  // 操作结果：销毁队列  {  delete[]elems; // 释放队列的存储空间  }  template <class ElemType>  int SeqQueue<ElemType>::GetLength() const  // 操作结果：返回队列中元素个数  {  return (rear - front + maxSize) % maxSize;  }  template<class ElemType>  bool SeqQueue<ElemType>::IsEmpty() const  // 操作结果：如队列为空，则返回true，否则返回false  {  return front == rear;  }  template<class ElemType>  void SeqQueue<ElemType>::Clear()  // 操作结果：清空队列  {  front = 0;  rear = 0;  }  template <class ElemType>  void SeqQueue<ElemType>::Traverse(const std::function<void(const ElemType&)>& Visit) const {  for (int i = front; i != rear; i = (i + 1) % maxSize) {  Visit(elems[i]);  }  if (!IsEmpty()) {  Visit(elems[rear]);  }  }  template<class ElemType>  Status SeqQueue<ElemType>::EnQueue(const ElemType e)  // 操作结果：将元素e追加到队尾,如成功则返加SUCCESS,如队列已满将返回OVER\_FLOW  {  if ((rear + 1) % maxSize == front) // 队列已满  return OVER\_FLOW;  else { // 操作成功  elems[rear] = e; // 将元素e追加到队尾  rear = (rear + 1) % maxSize;  return SUCCESS;  }  }  template<class ElemType>  Status SeqQueue<ElemType>::DelQueue(ElemType& e)  // 操作结果：如队列非空,删除队头元素,并用e返回队头元素,函数返回SUCCESS,否则  // 函数返回UNDER\_FLOW  {  if (IsEmpty()) // 队列空  return UNDER\_FLOW;  else { // 操作成功  e = elems[front]; // 用e返回队头元素  front = (front + 1) % maxSize;  return SUCCESS;  }  }  template<class ElemType>  Status SeqQueue<ElemType>::GetHead(ElemType& e) const  // 操作结果：如队列非空,用e返回队首元素,函数返回SUCCESS,否则函数返回UNDER\_FLOW  {  if (IsEmpty()) // 队列空  return UNDER\_FLOW;  else { // 队列非空,操作成功  e = elems[front]; // 用e返回队首元素  return SUCCESS;  }  }  template<class ElemType>  SeqQueue<ElemType>::SeqQueue(const SeqQueue<ElemType>& s)  // 操作结果：由队列s构造新队列--复制构造函数  {  maxSize = s.maxSize; // 队列的最大容量  elems = new ElemType[maxSize]; // 分配存储空间  front = s.front; // 复制队头指针  rear = s.rear; // 复制队尾指针  for (int i = front; i != rear; i = (i + 1) % maxSize) {  elems[i] = s.elems[i];  }  }  template<class ElemType>  SeqQueue<ElemType>& SeqQueue<ElemType>::operator =(const SeqQueue<ElemType>& s)  // 操作结果：将队列s赋值给当前队列--赋值语句重载  {  if (&s != this) {  delete[]elems; // 释放已有存储空间  maxSize = s.maxSize; // 队列的最大容量  elems = new ElemType[maxSize]; // 分配存储空间  front = s.front; // 复制队头指针  rear = s.rear; // 复制队尾指针  for (int i = front; i != rear; i = (i + 1) % maxSize) {  elems[i] = s.elems[i];  }  }  return \*this;  }  #endif // \_\_SQ\_QUEUE\_H\_\_ |
| Main.cpp |
| #include "Assistance.h" // 引入辅助软件包，可能包含一些通用工具函数  #include "SeqQueue.h" // 引入循环队列类的头文件  #include "SeqStack.h" // 引入顺序栈类的头文件  #include <SFML/Graphics.hpp>  #include <vector>  #include <iostream>  #include <algorithm> // 包含 std::random\_shuffle  #include <functional> // 包含 std::function  int main(void) {  sf::RenderWindow window(sf::VideoMode(800, 600), "T-shaped Railway Scheduling System");  // 设置字体  sf::Font font;  if (!font.loadFromFile("arial.ttf")) {  std::cerr << "Failed to load font file arial.ttf" << std::endl;  return -1;  }  // 创建菜单文本  sf::Text menuText("T-shaped Railway Scheduling System\n", font, 24);  menuText.setPosition(200, 100);  sf::Text option1("1. Scheduling System (Input/Output)\n", font, 24);  option1.setPosition(250, 150);  sf::Text option2("2. Random Sequence Generation\n", font, 24);  option2.setPosition(250, 200);  sf::Text option5("5. Exit Program\n", font, 24);  option5.setPosition(250, 250);  // 创建次级菜单文本  sf::Text stackOptionText("1. Use Stack as Assist Track\n", font, 24);  stackOptionText.setPosition(250, 150);  sf::Text queueOptionText("2. Use Queue as Assist Track\n", font, 24);  queueOptionText.setPosition(250, 200);  // 创建输入提示文本  sf::Text inputText("Enter the number of carriages (order from right to left): ", font, 24);  inputText.setPosition(100, 100);  sf::Text inputOrderText("Enter the order of carriages entering the stack: ", font, 24);  inputOrderText.setPosition(100, 150);  sf::Text inputEText("Enter e (e = 0 to exit): ", font, 24);  inputEText.setPosition(100, 200);  // 创建轨道状态文本  sf::Text mainTrackText("Main Track:", font, 24);  mainTrackText.setPosition(100, 450);  sf::Text assistStackText("Assist Track:", font, 24);  assistStackText.setPosition(100, 500);  sf::Text rightTrackText("Right Track:", font, 24);  rightTrackText.setPosition(100, 550);  // 创建变量  SeqQueue<int> qa; // 创建一个整型的队列 qa，表示主轨道  SeqQueue<int> qc; // 创建一个整型的队列 qc，表示右侧轨道  SeqStack<int> qb; // 创建一个整型的栈 qb，表示辅轨道  SeqQueue<int> qd; // 创建一个整型的队列 qd，用于替代栈作为辅轨道  int n = 0, x = 0, d = 0, No = 0; // 定义整型变量 n, x, d, No  int inst = 0; // 初始化入栈次数为0  int outst = 0; // 初始化出栈次数为0  int outqu = 0; // 初始化出队列次数为0  int inqu = 0; // 初始化入队列次数为0  int maxlen = 0; // 初始化辅轨道最大长度为0  // 用于存储用户输入的车厢编号  std::vector<int> inputCarriages;  int choice = 0;  int secondaryChoice = 0; // 用于存储栈或队列的选择  bool inputCompleted = false;  bool showMenu = true;  bool showSecondaryMenu = false; // 新增用于显示次级菜单的标志  while (window.isOpen()) {  sf::Event event;  while (window.pollEvent(event)) {  if (event.type == sf::Event::Closed) {  window.close();  }  else if (event.type == sf::Event::KeyPressed) {  if (event.key.code == sf::Keyboard::Num1 && showMenu) {  choice = 1;  showMenu = false;  showSecondaryMenu = true; // 显示次级菜单  }  else if (event.key.code == sf::Keyboard::Num2 && showMenu) {  choice = 2;  showMenu = false;  }  else if (event.key.code == sf::Keyboard::Num5 && showMenu) {  window.close();  }  else if (event.key.code == sf::Keyboard::Num1 && showSecondaryMenu) { // 选择栈作为辅轨道  secondaryChoice = 1;  showSecondaryMenu = false;  }  else if (event.key.code == sf::Keyboard::Num2 && showSecondaryMenu) { // 选择队列作为辅轨道  secondaryChoice = 2;  showSecondaryMenu = false;  }  else if (event.key.code == sf::Keyboard::Enter && !showMenu && !showSecondaryMenu) {  if (x == 0) {  inputCompleted = true;  }  else {  inputCarriages.push\_back(x);  }  x = 0;  }  else if (event.key.code >= sf::Keyboard::Num0 && event.key.code <= sf::Keyboard::Num9 && !showMenu && !showSecondaryMenu) {  x = x \* 10 + (event.key.code - sf::Keyboard::Num0);  }  else if (event.key.code == sf::Keyboard::BackSpace && !showMenu && !showSecondaryMenu) {  // 删除最后一位输入的数字  if (x > 0) {  x /= 10; // 去掉最后一位  }  }  }  }  window.clear(sf::Color::Black); // 将背景颜色改为黑色  if (showMenu) {  window.draw(menuText);  window.draw(option1);  window.draw(option2);  window.draw(option5);  }  else if (showSecondaryMenu) { // 新增次级菜单选择  window.draw(stackOptionText);  window.draw(queueOptionText);  }  else if (!inputCompleted && choice == 1) { // 只在选择1时显示输入提示  // 显示输入提示  window.draw(inputText);  window.draw(inputOrderText);  window.draw(inputEText);  // 显示当前输入的车厢编号  std::string carriagesStr = "Current input:";  for (int carriage : inputCarriages) {  carriagesStr += std::to\_string(carriage) + " ";  }  sf::Text carriagesText(carriagesStr, font, 24);  carriagesText.setPosition(450, 350);  window.draw(carriagesText);  // 显示当前输入的临时变量  sf::Text tempText(std::to\_string(x), font, 24);  tempText.setPosition(450, 400);  window.draw(tempText);  }  else {  // 处理调度逻辑  if (choice == 1) {  n = inputCarriages.size();  No = 1;  for (int carriage : inputCarriages) {  qa.EnQueue(carriage);  }  // 从 1 到 n 进行调度操作  for (int i = 1; i <= n; i++) {  if (secondaryChoice == 1) { // 使用栈作为辅助轨道  // 先检查辅轨道  if (!qb.IsEmpty()) {  qb.Top(d);  if ((d != No) && (qa.IsEmpty())) {  break;  }  else if (d == No) {  std::cout << "Carriage " << d << " from assist stack enters right track." << std::endl;  outst++; inqu++;  qc.EnQueue(d);  qb.Pop(d);  No++;  }  }  // 再检查主轨道  if (!qa.IsEmpty()) {  qa.GetHead(x);  if (No == x) {  std::cout << "Carriage " << x << " from left main track enters right main track." << std::endl;  outqu++; inqu++;  qc.EnQueue(x);  qa.DelQueue(x);  No++;  }  else if (No != x) {  while (No != x && !(qa.IsEmpty())) {  std::cout << "Carriage " << x << " from left main track enters assist stack." << std::endl;  d = x;  qb.Push(d);  qa.DelQueue(x);  qa.GetHead(x);  outqu++; inst++;  if (qb.GetLength() > maxlen) maxlen = qb.GetLength();  }  if (No == x) {  std::cout << "Carriage " << x << " from left main track enters right main track." << std::endl;  outqu++; inqu++;  qc.EnQueue(x);  qa.DelQueue(x);  No++;  continue;  }  }  }  }  else if (secondaryChoice == 2) { // 使用队列作为辅助轨道  // 先检查辅轨道  if (!qd.IsEmpty()) {  qd.GetHead(d);    if ((d != No) && (qa.IsEmpty())) {  break;  }  else if (d == No) {  std::cout << "Carriage " << d << " from assist queue enters right track." << std::endl;  outst++; inqu++;  qc.EnQueue(d);  qd.DelQueue(d);  No++;  }  }  // 再检查主轨道  if (!qa.IsEmpty()) {  qa.GetHead(x);    if (No == x) {  std::cout << "Carriage " << x << " from left main track enters right main track." << std::endl;  outqu++; inqu++;  qc.EnQueue(x);  qa.DelQueue(x);  No++;  }  // 如果队头元素不是当前需要调度的车厢  else if (No != x) {    while (No != x && !(qa.IsEmpty())) {  std::cout << "Carriage " << x << " from left main track enters assist queue." << std::endl;  d = x;  qd.EnQueue(x);  qa.DelQueue(x);  qa.GetHead(x);  outqu++; inst++;  if (qd.GetLength() > maxlen) maxlen = qd.GetLength();  }  // 如果队头元素是当前需要调度的车厢  if (No == x) {  std::cout << "Carriage " << x << " from left main track enters right main track." << std::endl;  outqu++; inqu++;  qc.EnQueue(x);  qa.DelQueue(x);  No++;  continue;  }  }  }  }  }  // 显示调度结果  if (qc.GetLength() == n) {  std::cout << "Scheduling completed." << std::endl;  std::cout << "Total number of push operations: " << inst << std::endl;  std::cout << "Total number of pop operations: " << outst << std::endl;  std::cout << "Total number of enqueue operations: " << outqu << std::endl;  std::cout << "Total number of dequeue operations: " << inqu << std::endl;  std::cout << "Maximum length of assist track: " << maxlen << std::endl;  }  else {  std::cout << "Scheduling cannot be completed." << std::endl;  }  }  else if (choice == 2) {  No = 1;  int maxcarriage = 5;  std::vector<int> nums(maxcarriage);  nums[0] = 1;  for (int k = 1; k < maxcarriage; k++) {  nums[k] = nums[k - 1] + 1;  }  std::random\_shuffle(nums.begin(), nums.end());  for (int j = 0; j < maxcarriage; j++) {  qa.EnQueue(nums[j]);  }  // 从 1 到 maxcarriage 进行调度操作  for (int j = 1; j <= maxcarriage; j++) {  // 先检查辅轨道  if (!qb.IsEmpty()) {  qb.Top(d);  if ((d != No) && (qa.IsEmpty())) {  break;  }  else if (d == No) {  std::cout << "Carriage " << d << " from assist stack enters right track." << std::endl;  outst++; inqu++;  qc.EnQueue(d);  qb.Pop(d);  No++;  }  }  // 再检查主轨道  if (!qa.IsEmpty()) {  qa.GetHead(x);  if (No == x) {  std::cout << "Carriage " << x << " from left main track enters right main track." << std::endl;  outqu++; inqu++;  qc.EnQueue(x);  qa.DelQueue(x);  No++;  }  else if (No != x) {  while (No != x && !(qa.IsEmpty())) {  std::cout << "Carriage " << x << " from left main track enters assist stack." << std::endl;  d = x;  qb.Push(d);  qa.DelQueue(x);  qa.GetHead(x);  outqu++; inst++;  if (qb.GetLength() > maxlen) maxlen = qb.GetLength();  }  if (No == x) {  std::cout << "Carriage " << x << " from left main track enters right main track." << std::endl;  outqu++; inqu++;  qc.EnQueue(x);  qa.DelQueue(x);  No++;  continue;  }  }  }  }  // 显示调度结果  if (qc.GetLength() == maxcarriage) {  std::cout << "Scheduling completed." << std::endl;  std::cout << "Total number of push operations: " << inst << std::endl;  std::cout << "Total number of pop operations: " << outst << std::endl;  std::cout << "Total number of enqueue operations: " << outqu << std::endl;  std::cout << "Total number of dequeue operations: " << inqu << std::endl;  std::cout << "Maximum length of assist stack: " << maxlen << std::endl;  }  else {  std::cout << "Scheduling cannot be completed." << std::endl;  }  }  // 显示当前轨道的状态  std::string mainTrackStr = "Main Track:";  qa.Traverse([&](int data) {  mainTrackStr += std::to\_string(data) + " ";  });  sf::Text mainTrack(mainTrackStr, font, 24);  mainTrack.setPosition(100, 450);  window.draw(mainTrack);  std::string assistTrackStr = "Assist Track:";  if (secondaryChoice == 1) {  qb.Traverse([&](int data) {  assistTrackStr += std::to\_string(data) + " ";  });  }  else if (secondaryChoice == 2) {  qd.Traverse([&](int data) {  assistTrackStr += std::to\_string(data) + " ";  });  }  sf::Text assistTrack(assistTrackStr, font, 24);  assistTrack.setPosition(100, 500);  window.draw(assistTrack);  std::string rightTrackStr = "Right Track:";  qc.Traverse([&](int data) {  rightTrackStr += std::to\_string(data) + " ";  });  sf::Text rightTrack(rightTrackStr, font, 24);  rightTrack.setPosition(100, 550);  window.draw(rightTrack);  // 重置输入状态  inputCarriages.clear();  inputCompleted = false;  showMenu = true;  showSecondaryMenu = false; // 重置次级菜单显示标志  qa.Clear();  qc.Clear();  qb.Clear();  qd.Clear();  inst = 0;  outst = 0;  outqu = 0;  inqu = 0;  maxlen = 0;  // 暂停以显示结果  sf::sleep(sf::seconds(2));  }  window.display();  }  return 0;  } |

下面对算法思想进行详细展开叙述：

1. 头文件Assistance.h辅助软件包

该头文件实现了一个基础的辅助软件包，其中包含了一些常见的功能，主要涉及输入输出、数组操作、错误处理和数据类型的操作等。通过合理地使用C++的标准库和一些自定义的函数与类，代码实现了若干实用的功能，可以方便地为其他项目提供支持。以下是对这段代码的总结，围绕功能的实现展开。

总体而言，该头文件实现了多个实用的功能模块，涵盖了输入输出、数组操作、交换、异常处理等方面的内容。这些功能模块被封装成了简单易用的函数，能够在实际开发中提高代码的复用性和可维护性。通过合理的命名和良好的结构设计，代码具有很高的可读性和扩展性，能够适应不同类型的需求。在实际应用中，可以方便地将这些辅助函数集成到更复杂的程序中，从而提高开发效率。

1. SeqStack类

该头文件定义了一个名为 SeqStack 的顺序栈模板类，它是一个线性数据结构，遵循后进先出（LIFO, Last In First Out）的原则。该模板类允许存储任何类型的元素，并提供了对栈的基本操作，如入栈、出栈、获取栈顶元素等。

顺序栈通过动态分配的数组来实现，其中 top 指针指示栈顶的位置，maxSize 定义了栈的最大容量，而 elems 是一个指向栈元素的指针。当创建一个新的 SeqStack 对象时，构造函数会根据提供的大小参数初始化栈，并为栈元素分配内存空间。此时栈是空的，top 被设置为 -1 表示没有元素。析构函数会在对象生命周期结束时释放栈占用的内存资源，防止内存泄漏。

对于栈的基本操作，GetLength() 方法返回当前栈中元素的数量top+1(因为 top 从 -1 开始计数)。IsEmpty() 检查栈是否为空，即 top 是否等于 -1。Clear() 方法将栈重置为空状态，即将 top 设回 -1。Traverse() 方法允许用户遍历栈中所有元素并执行特定操作，这通过传递给它的 std::function 回调函数完成。入栈操作 Push(e) 将新元素添加到栈顶；Top(e) 方法用于获取栈顶元素的值并将其赋给参数 e，但不移除该元素；Pop(e) 方法不仅返回栈顶元素的值给 e，还会实际移除该元素，即减少 top 的值。

1. SeqQueue类

该头文件实现了一个循环队列（SeqQueue）模板类，它允许以线性但循环的方式存储元素，并且提供了基本的队列操作。循环队列是一种特殊的队列类型，它的最后一个物理位置连接到第一个物理位置，形成一个环状结构，这有助于更高效地利用固定的内存空间。以下是该类的设计思路：

数据成员front 和 rear 分别作为队头和队尾指针，用于跟踪队列中的第一个和最后一个元素的位置。maxSize 表示队列的最大容量。elems 是一个动态分配的数组，用来实际存储队列元素。

构造函数初始化一个新的空队列，并根据提供的大小参数分配相应的内存空间。默认情况下，队列是空的，即 front 和 rear 都指向队列的第一个位置。析构函数负责释放队列所占用的内存资源，防止内存泄漏。复制构造函数创建现有队列的一个副本，确保两个队列互不影响。赋值运算符重载实现了深拷贝，同样保证了两个队列之间的独立性。

函数成员GetLength() 通过计算 (rear - front + maxSize) % maxSize 来确定当前队列中元素的数量；IsEmpty() 用于检查队列是否为空；Clear() 将队列重置为空状态。Traverse() 方法遍历队列中的所有元素；EnQueue(e) 在队尾插入新元素 e；DelQueue(e) 移除队首元素，并将其值赋给 e；GetHead(e) 获取队首元素的值并赋给 e，但不移除此元素。

（4）设计思路

初始化：在开始调度之前，程序已经初始化了四个队列/栈对象：qa（左主轨道）、qb（辅助栈）、qd（辅助队列）以及 qc（右主轨道）。同时，定义了一些计数器变量来跟踪操作次数（如入栈、出栈、入队、出队等）和辅助轨道的最大长度。

调度循环：代码通过一个 for 循环迭代从 1 到 n，代表对每个车厢进行处理。根据用户选择 (secondaryChoice) 决定是使用栈还是队列作为辅助轨道。

检查辅助轨道：如果选择了栈作为辅助轨道 (secondaryChoice == 1)，首先尝试从栈顶取出元素并与当前编号 No 比较。如果匹配，则将该车厢移动到右主轨道，并更新相关计数器。对于队列作为辅助轨道的情况 (secondaryChoice == 2)，逻辑相似，但操作的是队首元素。

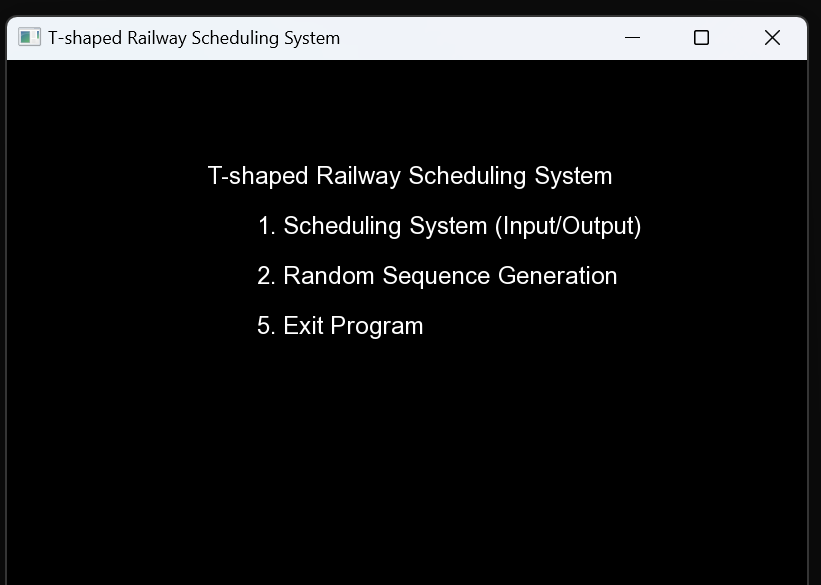
检查主轨道：接下来，检查左主轨道是否为空。如果不为空，则获取其头部元素 x 并与 No 比较。如果匹配，则直接将车厢移至右主轨道；如果不匹配，则继续检查下一个车厢，直到找到匹配的车厢或将所有不匹配的车厢移到辅助轨道上。

更新最大长度：在每次向辅助轨道添加新元素时，都会检查当前辅助轨道的长度是否超过记录的最大长度 maxlen，如果是则更新 maxlen。

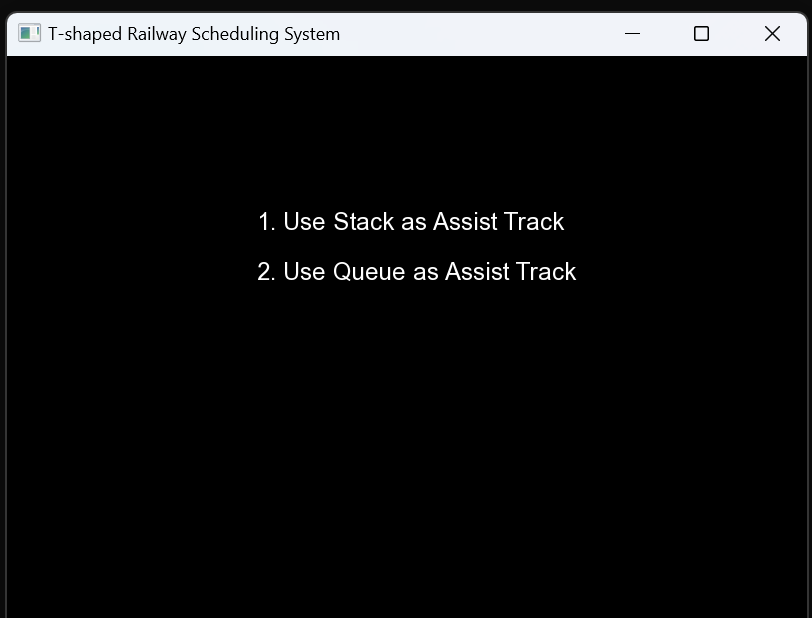
当所有车厢都被成功调度到右主轨道后，程序会输出调度完成的信息以及统计的操作次数和辅助轨道的最大长度。如果未能成功调度所有车厢，则输出调度失败的消息。

**2、实验结果**

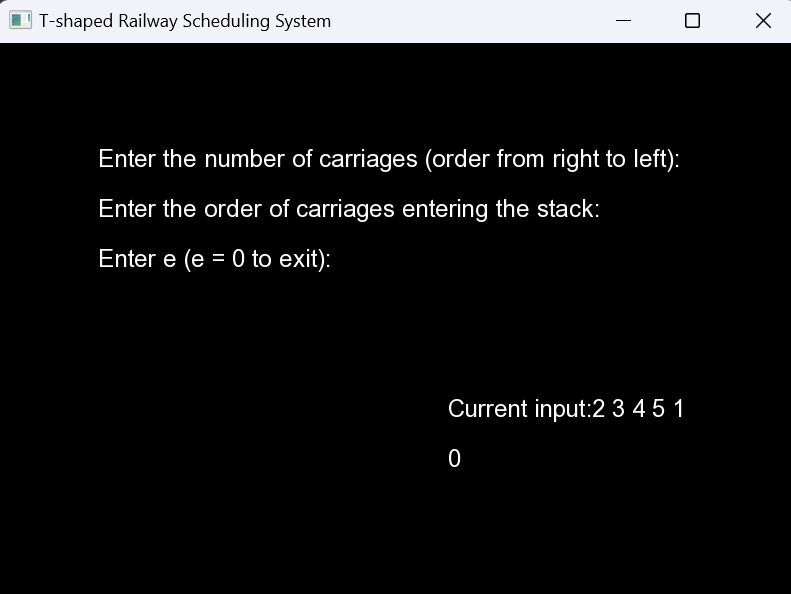
交互界面

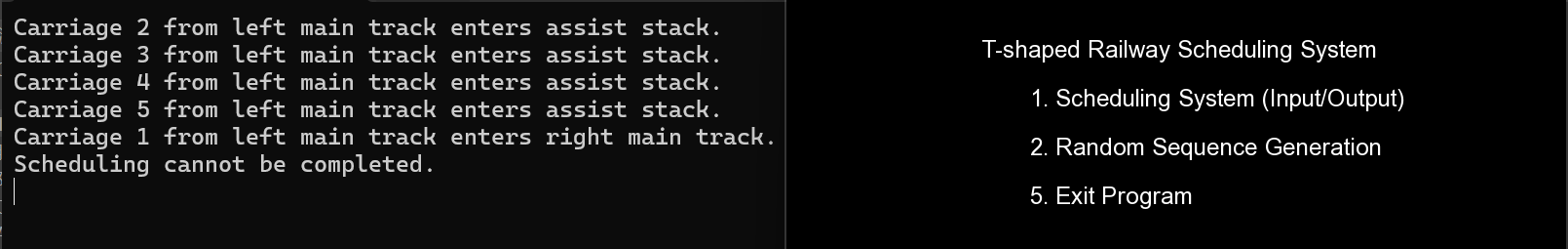


（1）输入1，进入调度系统。

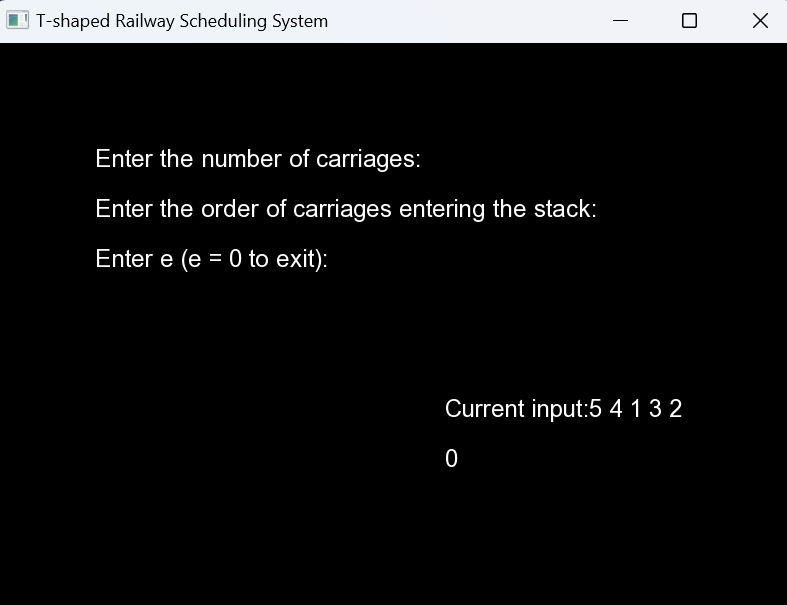


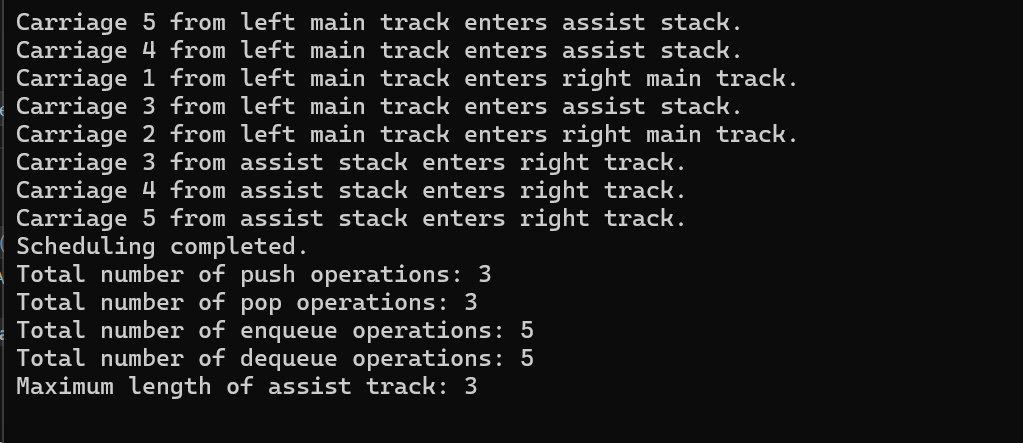
I输入1，使用栈作为辅助轨道。





车厢驶入顺序为2 3 4 5 1，无法实现按照1 2 3 4 5的顺序开出

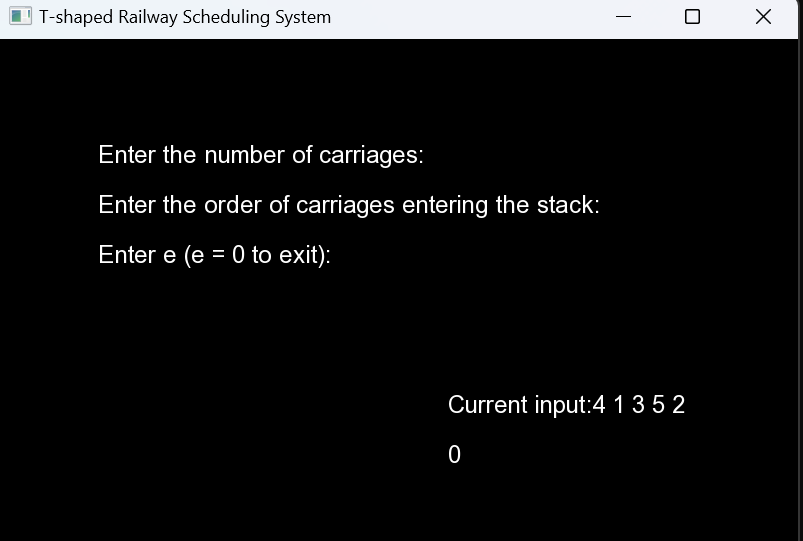


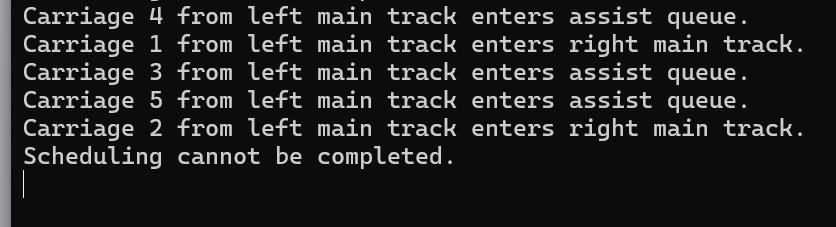


车厢驶入顺序为5 4 1 3 2，成功按照1 2 3 4 5的顺序驶出

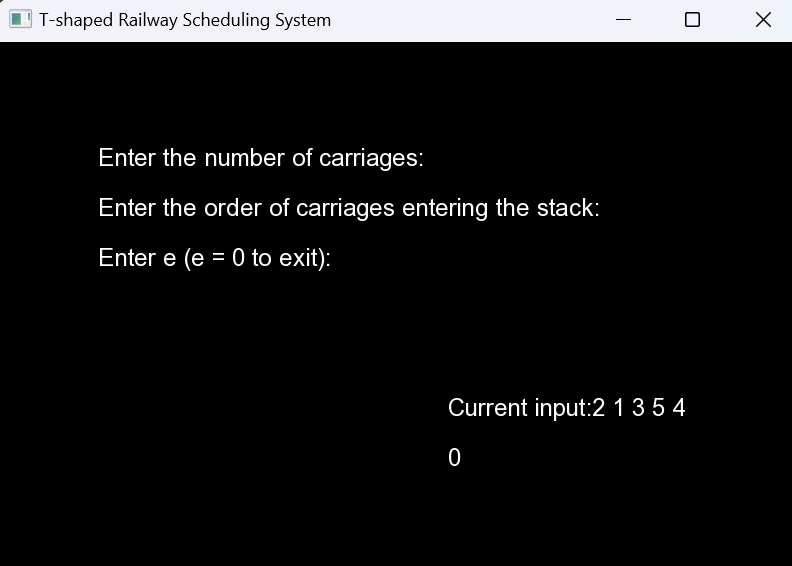
入栈次数和出栈次数均为3，入队列次数和出队列次数均为5，辅轨道最大长度为3。

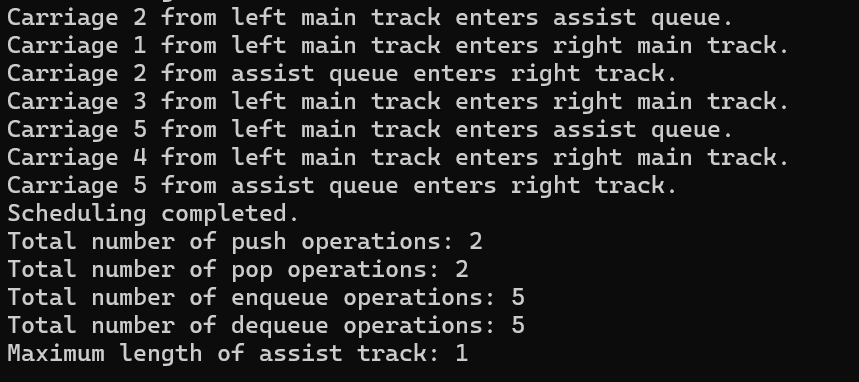
II输入2，使用队列作为辅助轨道





车厢驶入顺序为4 1 3 5 2，无法实现按照1 2 3 4 5的顺序开出





车厢驶入顺序为2 1 3 5 4，成功按照1 2 3 4 5的顺序驶出

入栈次数为2，出栈次数为0，出队列次数为7，入队列次数为5，辅轨道最大长度为1。

1. 输入2，随机生成100个初始序列，测试这些初始序列能否被成功调度，并输出



（3）输入5，退出程序。

**3、算法分析**

**时间复杂度**

(1)检查辅助轨道：

对于栈 (qb) 或队列 (qd) 辅助轨道的检查和可能的操作（如Top、Pop 或 GetHead、DelQueue）都是 O(1) 操作。

(2)检查主轨道：

获取头部元素 (GetHead) 是 O(1) 操作。如果当前编号 No 与头部元素不匹配，则需要将不匹配的元素移至辅助轨道。这可能导致一个内部 while 循环，最坏情况下会遍历整个 qa 队列，即 O(n) 操作，其中 n 是剩余未处理的车厢数量。

(3)更新最大长度：

更新辅助轨道的最大长度是一个简单的比较操作，时间复杂度为 O(1)。

(4)外部循环：

外部的 for 循环迭代次数是固定的 n 次，对应于从 1 到 n 的每个车厢。

最坏情况分析：

在最坏的情况下，对于每一个车厢 i (1 ≤ i ≤ n)，如果它不是当前需要调度的车厢 No，那么它可能会被临时移到辅助轨道上，然后再移回主轨道或直接移到右主轨道。这意味着每次内层 while 循环都可能执行最多 n-i 次，其中 i 是当前的车厢编号。

第一次迭代时，所有 n 个车厢都需要检查，所以是 O(n)。

第二次迭代时，剩下 n-1 个车厢需要检查，所以是 O(n-1)。

...

最后一次迭代时，只剩下 1 个车厢需要检查，所以是 O(1)。

因此，在最坏情况下，总的时间复杂度可以表示为 O(n + (n-1) + ... + 1)，这是一个等差数列求和问题，结果为 O(n\*(n+1)/2)，简化后为 O(n^2)。

平均情况分析：

平均情况下，由于车厢进入和离开辅助轨道的频率不一定达到最坏情况，实际性能可能会更好一些。然而，考虑到算法结构中存在潜在的双重循环（外层 for 和内层 while），即便是在平均情况下，时间复杂度也可能是接近 O(n^2) 的。

**空间复杂度**

主要空间消耗有：

(1)输入轨道 (qa)：存储所有 n 个车厢，因此需要 O(n) 的空间。

(2)输出轨道 (qc)：最终也需要存储所有 n 个车厢，所以同样需要 O(n) 的空间。

(3)辅助轨道：可以是栈 (qb) 或者队列 (qd)。在最坏情况下，所有的车厢都可能被临时移动到辅助轨道上，这意味着辅助轨道也可能需要 O(n) 的空间。

(4)辅助变量：包括用于计数的操作次数变量（如 inst, outst, inqu, outqu）和最大长度变量 (maxlen) 等，这些都需要常量级别的额外空间 O(1)。

综合考虑上述各个部分，我们可以得出以下结论：

整个程序的空间复杂度为 O(n)。这是因为无论选择栈还是队列作为辅助轨道，最坏情况下都可能需要额外的 O(n) 空间来存储所有车厢。此外，输入和输出轨道本身就占用了 O(n) 的空间。

**4、总结与心得**

（1）知识巩固与提升

此次铁路调度系统的编程练习，加深了对栈和队列这两种基本数据结构的理解，学会了如何根据具体的应用场景灵活运用这些数据结构。例如，在本项目中，我们选择了栈作为辅助轨道，充分利用其后进先出（LIFO, Last In First Out）的特点来帮助重新排列车厢顺序。这使我们意识到，不同数据结构各有其独特的特性和适用范围，理解这些特性对于解决实际问题至关重要。

此外，通过本次项目，我们还认识到理论知识与实际编码之间的差距。虽然在课堂上学习了栈和队列的定义及操作，但将这些概念转化为实际代码，并确保程序能够正确、高效地运行，是完全不同的挑战。这次实践让我更加清楚地认识到，掌握理论只是第一步，真正的能力在于能否将理论应用于实践中，解决问题并优化性能。

（2）编程技能提升

在代码实现过程中，提高了处理复杂逻辑和算法的能力。编写一个能够正确模拟铁路调度过程的程序并非易事，需要考虑多种因素，如车厢的进入顺序、辅助轨道的选择、以及如何有效地管理和追踪操作次数等。为了实现这一目标，我们不得不深入思考每个步骤，确保所有细节都被妥善处理。这不仅提升了我们对逻辑设计的理解，也增强了调试和优化代码的能力。

（3）团队协作与问题解决

在整个实验过程中，小组成员分工明确，各自负责项目的不同部分，如需求分析、算法设计、代码实现和测试验证等。这种明确的分工使我们能够高效地推进项目进度，同时也保证了每个环节的质量。当遇到困难时，团队成员之间积极讨论和交流，共同寻找解决方案。

1. 不足与改进方向

尽管项目成功实现了基本功能，但在某些情况下仍存在性能瓶颈，特别是最坏情况下的时间复杂度较高。例如，当车厢进入顺序非常不利时，可能会导致大量的入栈和出栈操作，进而影响整体效率。这提醒我们在未来的项目中，应更加注重算法优化，尽可能减少不必要的计算和存储开销。为此，可以考虑引入更高效的排序算法或数据结构，或者探索动态规划等高级算法来进一步提升性能。